

**Name: Nikhil Kumar**

**Roll: 1806055 , CSE 6th Sem**

### **Information Security Assignment 01**

---

**Q1: (a and b)**

#### **Linear Congruential Generator**

**code:**

$X_{n+1} = (aX_n + c) \bmod m$

```
def. linearCongruentialMethod(seed, multiplier, increment, modulus, totalRandomNumbers):
```

```
    randomNumbers = [0 for _ in range(totalRandomNumbers)]
```

```
    randomNumbers[0] = seed
```

```
    for idx in range(1, totalRandomNumbers):
```

```
        randomNumbers[idx] = (randomNumbers[idx-1]*multiplier + increment) % modulus
```

```
##### For Finding Period
```

```
for idx in range(1, len(randomNumbers)):
```

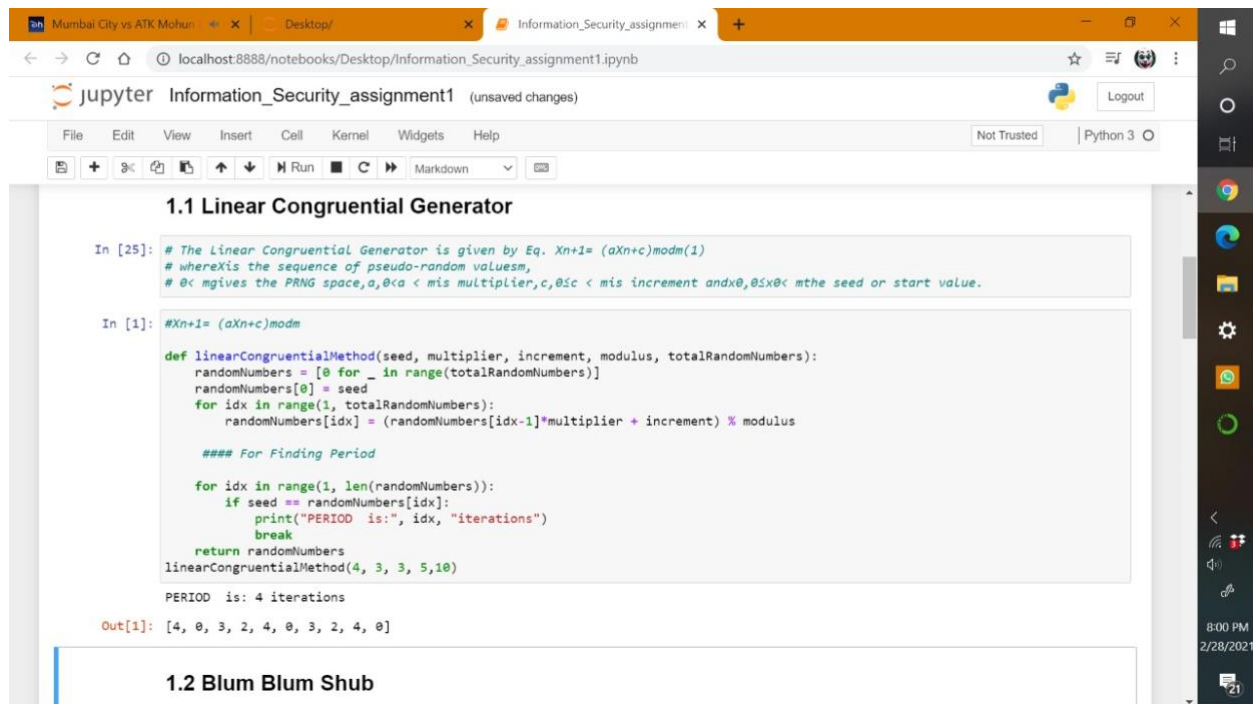
```
    if seed == randomNumbers[idx]:
```

```
        print("period is:", idx, "iterations")
```

```
        break
```

```
    return randomNumbers
```

linearCongruentialMethod(4, 3, 3, 5,10)



The screenshot shows a Jupyter Notebook interface with a browser window at the top displaying the URL `localhost:8888/notebooks/Desktop/Information_Security_assignment1.ipynb`. The notebook has a menu bar with File, Edit, View, Insert, Cell, Kernel, Widgets, and Help. Below the menu is a toolbar with icons for file operations, running, and markdown. The notebook content is divided into two sections: 1.1 Linear Congruential Generator and 1.2 Blum Blum Shub. In section 1.1, there is a code cell with a comment explaining the Linear Congruential Generator formula  $X_{n+1} = (aX_n + c) \bmod m$  and its parameters. Below the comment is a Python function `linearCongruentialMethod` that takes seed, multiplier, increment, modulus, and totalRandomNumbers as arguments. The function initializes a list of random numbers, sets the seed, and then iterates to generate random numbers using the LCG formula. It also includes a loop to find the period of the generator. The output of the function call `linearCongruentialMethod(4, 3, 3, 5,10)` is displayed as `Out[1]: [4, 0, 3, 2, 4, 0, 3, 2, 4, 0]`.

```
In [25]: # The Linear Congruential Generator is given by Eq.  $X_{n+1} = (aX_n + c) \bmod m$ 
# where  $X_i$  is the sequence of pseudo-random values,
#  $0 < m$  gives the PRNG space,  $a, 0 < a < m$  is multiplier,  $c, 0 \leq c < m$  is increment and  $x_0, 0 \leq x_0 < m$  the seed or start value.

In [1]: #  $X_{n+1} = (aX_n + c) \bmod m$ 

def linearCongruentialMethod(seed, multiplier, increment, modulus, totalRandomNumbers):
    randomNumbers = [0 for _ in range(totalRandomNumbers)]
    randomNumbers[0] = seed
    for idx in range(1, totalRandomNumbers):
        randomNumbers[idx] = (randomNumbers[idx-1]*multiplier + increment) % modulus

    ### For Finding Period

    for idx in range(1, len(randomNumbers)):
        if seed == randomNumbers[idx]:
            print("PERIOD is:", idx, "iterations")
            break
    return randomNumbers
linearCongruentialMethod(4, 3, 3, 5,10)

PERIOD is: 4 iterations

Out[1]: [4, 0, 3, 2, 4, 0, 3, 2, 4, 0]
```

output:

## Blum Blum Shub

Code:

```
##  $x_{n+1} = x_n^2 \bmod M$ 
```

```
import random
```

```
class BlumBlumShub:
```

```
    def __init__(self, length):
```

```
        self.length = length
```

```
        self.primes = self.generatePrimes(1000)
```

```
    def generatePrimes(self, number):
```

```
        primeNumbers = []
```

```
        for num in range(number):
```

```
            if self.isPrimeValue(num):
```

```
                primeNumbers.append(num)
```

```
return primeNumbers
```

```
def isPrimeValue(self, number):
```

```
for num in range(2, number):
```

```
if number % num == 0:
```

```
return False
```

```
return True
```

```
def getPrimes(self):
```

```
primeValues = []
```

```
while len(primeValues) < 2:
```

```
currentPrime = self.primes.pop()
```

```
if currentPrime % 4 == 3:
```

```
primeValues.append(currentPrime)
```

```
return primeValues
```

```
def setRandomSequence(self):
```

```
x = random.randrange(1000)
```

```
randomSequence = []
```

```
for _ in range(self.length):
```

```
x += 1
```

```
p, q = self.getPrimes()
```

```
m = p * q
```

```
z = (x**2) % m
```

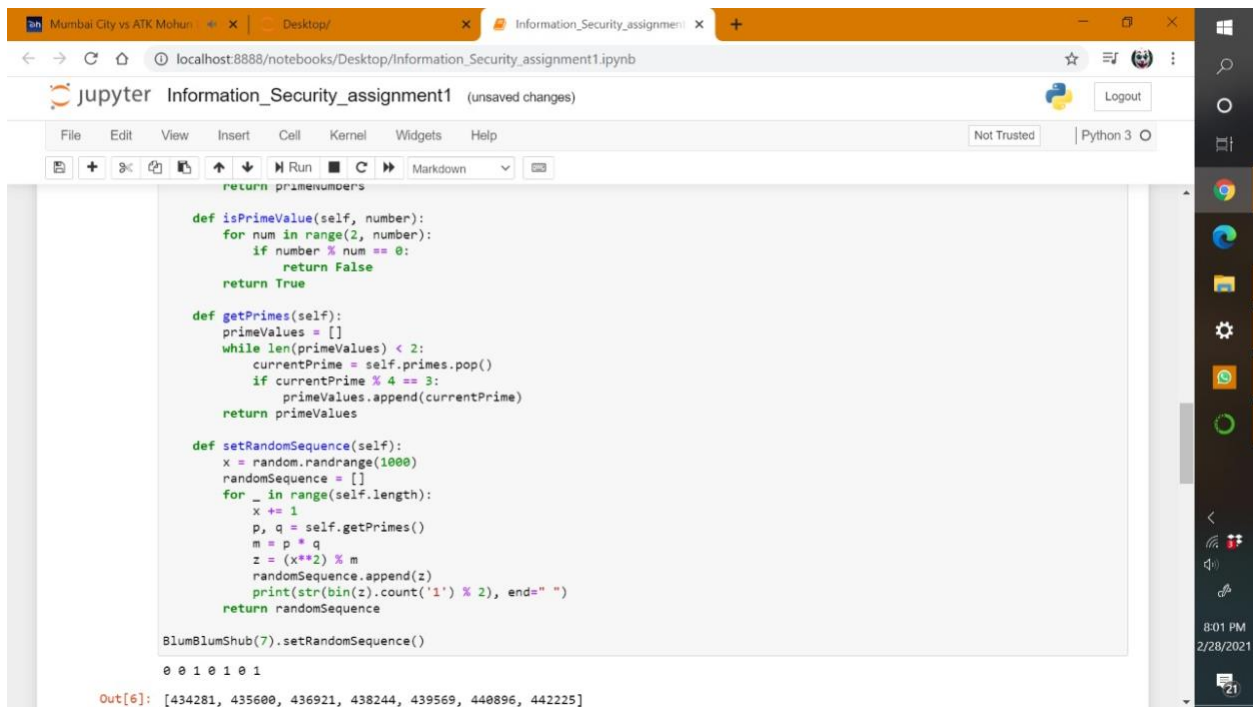
```
randomSequence.append(z)
```

```
print(str(bin(z).count('1') % 2), end=" ")
```

```
return randomSequence
```

```
BlumBlumShub(7).setRandomSequence()
```

output:



```
return primenumbers

def isPrimeValue(self, number):
    for num in range(2, number):
        if number % num == 0:
            return False
    return True

def getPrimes(self):
    primeValues = []
    while len(primeValues) < 2:
        currentPrime = self.primes.pop()
        if currentPrime % 4 == 3:
            primeValues.append(currentPrime)
    return primeValues

def setRandomSequence(self):
    x = random.randrange(1000)
    randomSequence = []
    for _ in range(self.length):
        x += 1
        p, q = self.getPrimes()
        m = p * q
        z = (x**2) % m
        randomSequence.append(z)
        print(str(bin(z).count('1') % 2), end=" ")
    return randomSequence

BlumBlumShub(7).setRandomSequence()

0 0 1 0 1 0 1

Out[6]: [434281, 435600, 436921, 438244, 439569, 440896, 442225]
```

## Linear Feedback Shift Register

Code:

```
def linearFeedBackShiftRegister(seed, postionOfTaps):
```

```
    shiftRegister, xor = seed, 0
```

```
    period = 0
```

```
    while True:
```

```
        for position in postionOfTaps:
```

```
            period += 1
```

```
            A = int(shiftRegister[len(shiftRegister)-position])
```

```
            B = int(shiftRegister[0])
```

```
            notA = 0 if A == 1 else 1
```

```
            notB = 0 if B == 1 else 1
```

```
            xor = A*notB + notA*B
```

```
            shiftRegister= shiftRegister[1:] + str(xor)
```

```
            print((shiftRegister, xor), end=" ")
```

```
linearFeedbackShiftRegister('01101000010', (9, 5, 2))
```

**output:**

[illegible]

## Q2: Solution

A random number generator needs to be secure against attack by an adversary who knows the algorithm and a (large) number of previously generated bits in order to be considered cryptographically secure. What this means is that someone with that information can't reconstruct any of the hidden internal state of the generator and give predictions of what the next bits produced will be with better than 50% accuracy. Normal pseudo-random number generators are generally not cryptographically secure, as reconstructing the internal state from previously output bits is generally trivial (often, the entire internal state is just the last N bits produced directly). Any random number generator without good statistical properties is also not cryptographically secure, as its output is at least partly predictable even without knowing the internal state.

Any good crypto system can be used as a cryptographically secure random number generator use

the crypto system to encrypt the output of a 'normal' random number generator. Since an adversary can't reconstruct the plaintext output of the normal random number generator, he can't attack it directly. This is a somewhat circular definition and begs the question of how you key the crypto system to keep it secure, which is a whole other problem.

**Name: Nikhil kumar**

**Roll no:1806055**